EXHIBIT E

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```java
/*
 * Copyright (C) 2008 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package java.util;

/**
 * This is a near duplicate of {@link TimSort}, modified for use with
 * arrays of objects that implement {@link Comparable}, instead of using
 * explicit comparators.
 *
 * <p>If you are using an optimizing VM, you may find that ComparableTimSort
 * offers no performance benefit over TimSort in conjunction with a
 * comparator that simply returns {@code ((Comparable)first).compareTo(Second)}.
 * If this is the case, you are better off deleting ComparableTimSort to
 * eliminate the code duplication.  (See Arrays.java for details.)
 */
class ComparableTimSort {
    /**
     * This is the minimum sized sequence that will be merged.  Shorter
     * sequences will be lengthened by calling binarySort.  If the entire
     * array is less than this length, no merges will be performed.
     *
     * This constant should be a power of two.  It was 64 in Tim Peter's C
     * implementation, but 32 was empirically determined to work better in
     * this implementation.  In the unlikely event that you set this constant
     * to be a number that's not a power of two, you'll need to change the
     * {@link #minRunLength} computation.
     *
     * If you decrease this constant, you must change the stackLen
     * computation in the TimSort constructor, or you risk an
     * ArrayOutOfBounds exception.  See listsort.txt for a discussion
     * of the minimum stack length required as a function of the length
     * of the array being sorted and the minimum merge sequence length.
     */
    private static final int MIN_MERGE = 32;

    /**
     * The array being sorted.
     */
```

```
    private final Object[] a;

    /**
     * When we get into galloping mode, we stay there until both runs win less
     * often than MIN_GALLOP consecutive times.
     */
    private static final int  MIN_GALLOP = 7;

    /**
     * This controls when we get *into* galloping mode.  It is initialized
     * to MIN_GALLOP.  The mergeLo and mergeHi methods nudge it higher for
     * random data, and lower for highly structured data.
     */
    private int minGallop = MIN_GALLOP;

    /**
     * Maximum initial size of tmp array, which is used for merging.  The array
     * can grow to accommodate demand.
     *
     * Unlike Tim's original C version, we do not allocate this much storage
     * when sorting smaller arrays.  This change was required for performance.
     */
    private static final int INITIAL_TMP_STORAGE_LENGTH = 256;

    /**
     * Temp storage for merges.
     */
    private Object[] tmp;

    /**
     * A stack of pending runs yet to be merged.  Run i starts at
     * address base[i] and extends for len[i] elements.  It's always
     * true (so long as the indices are in bounds) that:
     *
     *     runBase[i] + runLen[i] == runBase[i + 1]
     *
     * so we could cut the storage for this, but it's a minor amount,
     * and keeping all the info explicit simplifies the code.
     */
    private int stackSize = 0;  // Number of pending runs on stack
    private final int[] runBase;
    private final int[] runLen;

    /**
     * Asserts have been placed in if-statements for performace. To enable them,
     * set this field to true and enable them in VM with a command line flag.
     * If you modify this class, please do test the asserts!
     */
    private static final boolean DEBUG = false;

    /**
     * Creates a TimSort instance to maintain the state of an ongoing
```

```
     sort.
     *
     * @param a the array to be sorted
     */
    private ComparableTimSort(Object[] a) {
        this.a = a;

        // Allocate temp storage (which may be increased later if
        necessary)
        int len = a.length;
        @SuppressWarnings({"unchecked", "UnnecessaryLocalVariable"})
        Object[] newArray = new Object[len < 2 *
        INITIAL_TMP_STORAGE_LENGTH ?
                                       len >>> 1 :
                                       INITIAL_TMP_STORAGE_LENGTH];
        tmp = newArray;

        /*
         * Allocate runs-to-be-merged stack (which cannot be expanded).
         The
         * stack length requirements are described in listsort.txt.  The
         C
         * version always uses the same stack length (85), but this was
         * measured to be too expensive when sorting "mid-sized" arrays
         (e.g.,
         * 100 elements) in Java.  Therefore, we use smaller (but
         sufficiently
         * large) stack lengths for smaller arrays.  The "magic numbers"
         in the
         * computation below must be changed if MIN_MERGE is decreased.
         See
         * the MIN_MERGE declaration above for more information.
         */
        int stackLen = (len <    120  ?  5 :
                        len <   1542  ? 10 :
                        len < 119151  ? 19 : 40);
        runBase = new int[stackLen];
        runLen = new int[stackLen];
    }

    /*
     * The next two methods (which are package private and static)
     constitute
     * the entire API of this class.  Each of these methods obeys the
     contract
     * of the public method with the same signature in java.util.Arrays.
     */

    static void sort(Object[] a) {
          sort(a, 0, a.length);
    }

    static void sort(Object[] a, int lo, int hi) {
        rangeCheck(a.length, lo, hi);
        int nRemaining  = hi - lo;
        if (nRemaining < 2)
            return;  // Arrays of size 0 and 1 are always sorted

        // If array is small, do a "mini-TimSort" with no merges
        if (nRemaining < MIN_MERGE) {
```

```
            int initRunLen = countRunAndMakeAscending(a, lo, hi);
            binarySort(a, lo, hi, lo + initRunLen);
            return;
        }

        /**
         * March over the array once, left to right, finding natural runs,
         * extending short natural runs to minRun elements, and merging runs
         * to maintain stack invariant.
         */
        ComparableTimSort ts = new ComparableTimSort(a);
        int minRun = minRunLength(nRemaining);
        do {
            // Identify next run
            int runLen = countRunAndMakeAscending(a, lo, hi);

            // If run is short, extend to min(minRun, nRemaining)
            if (runLen < minRun) {
                int force = nRemaining <= minRun ? nRemaining : minRun;
                binarySort(a, lo, lo + force, lo + runLen);
                runLen = force;
            }

            // Push run onto pending-run stack, and maybe merge
            ts.pushRun(lo, runLen);
            ts.mergeCollapse();

            // Advance to find next run
            lo += runLen;
            nRemaining -= runLen;
        } while (nRemaining != 0);

        // Merge all remaining runs to complete sort
        if (DEBUG) assert lo == hi;
        ts.mergeForceCollapse();
        if (DEBUG) assert ts.stackSize == 1;
    }

    /**
     * Sorts the specified portion of the specified array using a binary
     * insertion sort.  This is the best method for sorting small numbers
     * of elements.  It requires O(n log n) compares, but O(n^2) data
     * movement (worst case).
     *
     * If the initial part of the specified range is already sorted,
     * this method can take advantage of it: the method assumes that the
     * elements from index {@code lo}, inclusive, to {@code start},
     * exclusive are already sorted.
     *
     * @param a the array in which a range is to be sorted
     * @param lo the index of the first element in the range to be sorted
     * @param hi the index after the last element in the range to be sorted
     * @param start the index of the first element in the range that is
     *        not already known to be sorted (@code lo <= start <= hi}
     */
```

```
    @SuppressWarnings("fallthrough")
    private static void binarySort(Object[] a, int lo, int hi, int start) {
        if (DEBUG) assert lo <= start && start <= hi;
        if (start == lo)
            start++;
        for ( ; start < hi; start++) {
            @SuppressWarnings("unchecked")
            Comparable<Object> pivot = (Comparable) a[start];

            // Set left (and right) to the index where a[start] (pivot) belongs
            int left = lo;
            int right = start;
            if (DEBUG) assert left <= right;
            /*
             * Invariants:
             *   pivot >= all in [lo, left).
             *   pivot <  all in [right, start).
             */
            while (left < right) {
                int mid = (left + right) >>> 1;
                if (pivot.compareTo(a[mid]) < 0)
                    right = mid;
                else
                    left = mid + 1;
            }
            if (DEBUG) assert left == right;

            /*
             * The invariants still hold: pivot >= all in [lo, left) and
             * pivot < all in [left, start), so pivot belongs at left. Note
             * that if there are elements equal to pivot, left points to the
             * first slot after them -- that's why this sort is stable.
             * Slide elements over to make room to make room for pivot.
             */
            int n = start - left;  // The number of elements to move
            // Switch is just an optimization for arraycopy in default case
            switch(n) {
                case 2:  a[left + 2] = a[left + 1];
                case 1:  a[left + 1] = a[left];
                         break;
                default: System.arraycopy(a, left, a, left + 1, n);
            }
            a[left] = pivot;
        }
    }

    /**
     * Returns the length of the run beginning at the specified position in
     * the specified array and reverses the run if it is descending (ensuring
     * that the run will always be ascending when the method returns).
     *
     * A run is the longest ascending sequence with:
     *
```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
     *    a[lo] <= a[lo + 1] <= a[lo + 2] <= ...
     *
     * or the longest descending sequence with:
     *
     *    a[lo] >  a[lo + 1] >  a[lo + 2] >  ...
     *
     * For its intended use in a stable mergesort, the strictness of the
     * definition of "descending" is needed so that the call can safely
     * reverse a descending sequence without violating stability.
     *
     * @param a the array in which a run is to be counted and possibly
     reversed
     * @param lo index of the first element in the run
     * @param hi index after the last element that may be contained in
     the run.
              It is required that @code{lo < hi}.
     * @return  the length of the run beginning at the specified
     position in
     *          the specified array
     */
    @SuppressWarnings("unchecked")
    private static int countRunAndMakeAscending(Object[] a, int lo, int
    hi) {
        if (DEBUG) assert lo < hi;
        int runHi = lo + 1;
        if (runHi == hi)
            return 1;

        // Find end of run, and reverse range if descending
        if (((Comparable) a[runHi++]).compareTo(a[lo]) < 0) { //
        Descending
            while(runHi < hi && ((Comparable)
            a[runHi]).compareTo(a[runHi - 1]) < 0)
                runHi++;
            reverseRange(a, lo, runHi);
        } else {                              // Ascending
            while (runHi < hi && ((Comparable)
            a[runHi]).compareTo(a[runHi - 1]) >= 0)
                runHi++;
        }

        return runHi - lo;
    }

    /**
     * Reverse the specified range of the specified array.
     *
     * @param a the array in which a range is to be reversed
     * @param lo the index of the first element in the range to be
     reversed
     * @param hi the index after the last element in the range to be
     reversed
     */
    private static void reverseRange(Object[] a, int lo, int hi) {
        hi--;
        while (lo < hi) {
            Object t = a[lo];
            a[lo++] = a[hi];
            a[hi--] = t;
        }
```

```
    }

    /**
     * Returns the minimum acceptable run length for an array of the
     specified
     * length. Natural runs shorter than this will be extended with
     * {@link #binarySort}.
     *
     * Roughly speaking, the computation is:
     *
     *  If n < MIN_MERGE, return n (it's too small to bother with fancy
     stuff).
     *  Else if n is an exact power of 2, return MIN_MERGE/2.
     *  Else return an int k, MIN_MERGE/2 <= k <= MIN_MERGE, such that
     n/k
     *   is close to, but strictly less than, an exact power of 2.
     *
     * For the rationale, see listsort.txt.
     *
     * @param n the length of the array to be sorted
     * @return the length of the minimum run to be merged
     */
    private static int minRunLength(int n) {
        if (DEBUG) assert n >= 0;
        int r = 0;      // Becomes 1 if any 1 bits are shifted off
        while (n >= MIN_MERGE) {
            r |= (n & 1);
            n >>= 1;
        }
        return n + r;
    }

    /**
     * Pushes the specified run onto the pending-run stack.
     *
     * @param runBase index of the first element in the run
     * @param runLen  the number of elements in the run
     */
    private void pushRun(int runBase, int runLen) {
        this.runBase[stackSize] = runBase;
        this.runLen[stackSize] = runLen;
        stackSize++;
    }

    /**
     * Examines the stack of runs waiting to be merged and merges
     adjacent runs
     * until the stack invariants are reestablished:
     *
     *     1. runLen[i - 3] > runLen[i - 2] + runLen[i - 1]
     *     2. runLen[i - 2] > runLen[i - 1]
     *
     * This method is called each time a new run is pushed onto the
     stack,
     * so the invariants are guaranteed to hold for i < stackSize upon
     * entry to the method.
     */
    private void mergeCollapse() {
        while (stackSize > 1) {
            int n = stackSize - 2;
```

```
            if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
                if (runLen[n - 1] < runLen[n + 1])
                    n--;
                mergeAt(n);
            } else if (runLen[n] <= runLen[n + 1]) {
                mergeAt(n);
            } else {
                break; // Invariant is established
            }
        }
    }

    /**
     * Merges all runs on the stack until only one remains.  This method
     is
     * called once, to complete the sort.
     */
    private void mergeForceCollapse() {
        while (stackSize > 1) {
            int n = stackSize - 2;
            if (n > 0 && runLen[n - 1] < runLen[n + 1])
                n--;
            mergeAt(n);
        }
    }

    /**
     * Merges the two runs at stack indices i and i+1.  Run i must be
     * the penultimate or antepenultimate run on the stack.  In other
     words,
     * i must be equal to stackSize-2 or stackSize-3.
     *
     * @param i stack index of the first of the two runs to merge
     */
    @SuppressWarnings("unchecked")
    private void mergeAt(int i) {
        if (DEBUG) assert stackSize >= 2;
        if (DEBUG) assert i >= 0;
        if (DEBUG) assert i == stackSize - 2 || i == stackSize - 3;

        int base1 = runBase[i];
        int len1 = runLen[i];
        int base2 = runBase[i + 1];
        int len2 = runLen[i + 1];
        if (DEBUG) assert len1 > 0 && len2 > 0;
        if (DEBUG) assert base1 + len1 == base2;

        /*
         * Record the length of the combined runs; if i is the 3rd-last
         * run now, also slide over the last run (which isn't involved
         * in this merge).  The current run (i+1) goes away in any case.
         */
        runLen[i] = len1 + len2;
        if (i == stackSize - 3) {
            runBase[i + 1] = runBase[i + 2];
            runLen[i + 1] = runLen[i + 2];
        }
        stackSize--;

        /*
```

```
         * Find where the first element of run2 goes in run1. Prior elements
         * in run1 can be ignored (because they're already in place).
         */
        int k = gallopRight((Comparable<Object>) a[base2], a, base1, len1, 0);
        if (DEBUG) assert k >= 0;
        base1 += k;
        len1 -= k;
        if (len1 == 0)
            return;

        /*
         * Find where the last element of run1 goes in run2. Subsequent elements
         * in run2 can be ignored (because they're already in place).
         */
        len2 = gallopLeft((Comparable<Object>) a[base1 + len1 - 1], a,
                base2, len2, len2 - 1);
        if (DEBUG) assert len2 >= 0;
        if (len2 == 0)
            return;

        // Merge remaining runs, using tmp array with min(len1, len2) elements
        if (len1 <= len2)
            mergeLo(base1, len1, base2, len2);
        else
            mergeHi(base1, len1, base2, len2);
    }

    /**
     * Locates the position at which to insert the specified key into the
     * specified sorted range; if the range contains an element equal to key,
     * returns the index of the leftmost equal element.
     *
     * @param key the key whose insertion point to search for
     * @param a the array in which to search
     * @param base the index of the first element in the range
     * @param len the length of the range; must be > 0
     * @param hint the index at which to begin the search, 0 <= hint < n.
     *     The closer hint is to the result, the faster this method will run.
     * @return the int k,  0 <= k <= n such that a[b + k - 1] < key <= a[b + k],
     *    pretending that a[b - 1] is minus infinity and a[b + n] is infinity.
     *    In other words, key belongs at index b + k; or in other words,
     *    the first k elements of a should precede key, and the last n - k
     *    should follow it.
     */
    private static int gallopLeft(Comparable<Object> key, Object[] a,
            int base, int len, int hint) {
        if (DEBUG) assert len > 0 && hint >= 0 && hint < len;

        int lastOfs = 0;
```

```
        int ofs = 1;
        if (key.compareTo(a[base + hint]) > 0) {
            // Gallop right until a[base+hint+lastOfs] < key <= a[base+hint+ofs]
            int maxOfs = len - hint;
            while (ofs < maxOfs && key.compareTo(a[base + hint + ofs]) > 0) {
                lastOfs = ofs;
                ofs = (ofs << 1) + 1;
                if (ofs <= 0)   // int overflow
                    ofs = maxOfs;
            }
            if (ofs > maxOfs)
                ofs = maxOfs;

            // Make offsets relative to base
            lastOfs += hint;
            ofs += hint;
        } else { // key <= a[base + hint]
            // Gallop left until a[base+hint-ofs] < key <= a[base+hint-lastOfs]
            final int maxOfs = hint + 1;
            while (ofs < maxOfs && key.compareTo(a[base + hint - ofs]) <= 0) {
                lastOfs = ofs;
                ofs = (ofs << 1) + 1;
                if (ofs <= 0)   // int overflow
                    ofs = maxOfs;
            }
            if (ofs > maxOfs)
                ofs = maxOfs;

            // Make offsets relative to base
            int tmp = lastOfs;
            lastOfs = hint - ofs;
            ofs = hint - tmp;
        }
        if (DEBUG) assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;

        /*
         * Now a[base+lastOfs] < key <= a[base+ofs], so key belongs somewhere
         * to the right of lastOfs but no farther right than ofs.  Do a binary
         * search, with invariant a[base + lastOfs - 1] < key <= a[base + ofs].
         */
        lastOfs++;
        while (lastOfs < ofs) {
            int m = lastOfs + ((ofs - lastOfs) >>> 1);

            if (key.compareTo(a[base + m]) > 0)
                lastOfs = m + 1;  // a[base + m] < key
            else
                ofs = m;          // key <= a[base + m]
        }
        if (DEBUG) assert lastOfs == ofs;    // so a[base + ofs - 1] < key <= a[base + ofs]
        return ofs;
    }
```

```java

    /**
     * Like gallopLeft, except that if the range contains an element equal to
     * key, gallopRight returns the index after the rightmost equal element.
     *
     * @param key the key whose insertion point to search for
     * @param a the array in which to search
     * @param base the index of the first element in the range
     * @param len the length of the range; must be > 0
     * @param hint the index at which to begin the search, 0 <= hint < n.
     *     The closer hint is to the result, the faster this method will run.
     * @return the int k,  0 <= k <= n such that a[b + k - 1] <= key < a[b + k]
     */
    private static int gallopRight(Comparable<Object> key, Object[] a,
            int base, int len, int hint) {
        if (DEBUG) assert len > 0 && hint >= 0 && hint < len;

        int ofs = 1;
        int lastOfs = 0;
        if (key.compareTo(a[base + hint]) < 0) {
            // Gallop left until a[b+hint - ofs] <= key < a[b+hint - lastOfs]
            int maxOfs = hint + 1;
            while (ofs < maxOfs && key.compareTo(a[base + hint - ofs]) < 0) {
                lastOfs = ofs;
                ofs = (ofs << 1) + 1;
                if (ofs <= 0)   // int overflow
                    ofs = maxOfs;
            }
            if (ofs > maxOfs)
                ofs = maxOfs;

            // Make offsets relative to b
            int tmp = lastOfs;
            lastOfs = hint - ofs;
            ofs = hint - tmp;
        } else { // a[b + hint] <= key
            // Gallop right until a[b+hint + lastOfs] <= key < a[b+hint + ofs]
            int maxOfs = len - hint;
            while (ofs < maxOfs && key.compareTo(a[base + hint + ofs]) >= 0) {
                lastOfs = ofs;
                ofs = (ofs << 1) + 1;
                if (ofs <= 0)   // int overflow
                    ofs = maxOfs;
            }
            if (ofs > maxOfs)
                ofs = maxOfs;

            // Make offsets relative to b
            lastOfs += hint;
            ofs += hint;
        }
```

```java
        if (DEBUG) assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;

        /*
         * Now a[b + lastOfs] <= key < a[b + ofs], so key belongs somewhere to
         * the right of lastOfs but no farther right than ofs.  Do a binary
         * search, with invariant a[b + lastOfs - 1] <= key < a[b + ofs].
         */
        lastOfs++;
        while (lastOfs < ofs) {
            int m = lastOfs + ((ofs - lastOfs) >>> 1);

            if (key.compareTo(a[base + m]) < 0)
                ofs = m;          // key < a[b + m]
            else
                lastOfs = m + 1;  // a[b + m] <= key
        }
        if (DEBUG) assert lastOfs == ofs;    // so a[b + ofs - 1] <= key < a[b + ofs]
        return ofs;
    }

    /**
     * Merges two adjacent runs in place, in a stable fashion.  The first
     * element of the first run must be greater than the first element of the
     * second run (a[base1] > a[base2]), and the last element of the first run
     * (a[base1 + len1-1]) must be greater than all elements of the second run.
     *
     * For performance, this method should be called only when len1 <= len2;
     * its twin, mergeHi should be called if len1 >= len2.  (Either method
     * may be called if len1 == len2.)
     *
     * @param base1 index of first element in first run to be merged
     * @param len1  length of first run to be merged (must be > 0)
     * @param base2 index of first element in second run to be merged
     *        (must be aBase + aLen)
     * @param len2  length of second run to be merged (must be > 0)
     */
    @SuppressWarnings("unchecked")
    private void mergeLo(int base1, int len1, int base2, int len2) {
        if (DEBUG) assert len1 > 0 && len2 > 0 && base1 + len1 == base2;

        // Copy first run into temp array
        Object[] a = this.a; // For performance
        Object[] tmp = ensureCapacity(len1);
        System.arraycopy(a, base1, tmp, 0, len1);

        int cursor1 = 0;       // Indexes into tmp array
        int cursor2 = base2;   // Indexes int a
        int dest = base1;      // Indexes int a

        // Move first element of second run and deal with degenerate
```

```
        cases
        a[dest++] = a[cursor2++];
        if (--len2 == 0) {
            System.arraycopy(tmp, cursor1, a, dest, len1);
            return;
        }
        if (len1 == 1) {
            System.arraycopy(a, cursor2, a, dest, len2);
            a[dest + len2] = tmp[cursor1]; // Last elt of run 1 to end
            of merge
            return;
        }

        int minGallop = this.minGallop;  // Use local variable for
        performance
    outer:
        while (true) {
            int count1 = 0; // Number of times in a row that first run
            won
            int count2 = 0; // Number of times in a row that second run
            won

            /*
             * Do the straightforward thing until (if ever) one run
             starts
             * winning consistently.
             */
            do {
                if (DEBUG) assert len1 > 1 && len2 > 0;
                if (((Comparable) a[cursor2]).compareTo(tmp[cursor1]) <
                0) {
                    a[dest++] = a[cursor2++];
                    count2++;
                    count1 = 0;
                    if (--len2 == 0)
                        break outer;
                } else {
                    a[dest++] = tmp[cursor1++];
                    count1++;
                    count2 = 0;
                    if (--len1 == 1)
                        break outer;
                }
            } while ((count1 | count2) < minGallop);

            /*
             * One run is winning so consistently that galloping may be
             a
             * huge win. So try that, and continue galloping until (if
             ever)
             * neither run appears to be winning consistently anymore.
             */
            do {
                if (DEBUG) assert len1 > 1 && len2 > 0;
                count1 = gallopRight((Comparable) a[cursor2], tmp,
                cursor1, len1, 0);
                if (count1 != 0) {
                    System.arraycopy(tmp, cursor1, a, dest, count1);
                    dest += count1;
                    cursor1 += count1;
```

```
                    len1 -= count1;
                    if (len1 <= 1)  // len1 == 1 || len1 == 0
                        break outer;
                }
                a[dest++] = a[cursor2++];
                if (--len2 == 0)
                    break outer;

                count2 = gallopLeft((Comparable) tmp[cursor1], a,
                cursor2, len2, 0);
                if (count2 != 0) {
                    System.arraycopy(a, cursor2, a, dest, count2);
                    dest += count2;
                    cursor2 += count2;
                    len2 -= count2;
                    if (len2 == 0)
                        break outer;
                }
                a[dest++] = tmp[cursor1++];
                if (--len1 == 1)
                    break outer;
                minGallop--;
            } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
            if (minGallop < 0)
                minGallop = 0;
            minGallop += 2;  // Penalize for leaving gallop mode
        }  // End of "outer" loop
        this.minGallop = minGallop < 1 ? 1 : minGallop;  // Write back
        to field

        if (len1 == 1) {
            if (DEBUG) assert len2 > 0;
            System.arraycopy(a, cursor2, a, dest, len2);
            a[dest + len2] = tmp[cursor1]; //  Last elt of run 1 to end
            of merge
        } else if (len1 == 0) {
            throw new IllegalArgumentException(
                "Comparison method violates its general contract!");
        } else {
            if (DEBUG) assert len2 == 0;
            if (DEBUG) assert len1 > 1;
            System.arraycopy(tmp, cursor1, a, dest, len1);
        }
    }

    /**
     * Like mergeLo, except that this method should be called only if
     * len1 >= len2; mergeLo should be called if len1 <= len2.  (Either
     method
     * may be called if len1 == len2.)
     *
     * @param base1 index of first element in first run to be merged
     * @param len1  length of first run to be merged (must be > 0)
     * @param base2 index of first element in second run to be merged
     *        (must be aBase + aLen)
     * @param len2  length of second run to be merged (must be > 0)
     */
    @SuppressWarnings("unchecked")
    private void mergeHi(int base1, int len1, int base2, int len2) {
        if (DEBUG) assert len1 > 0 && len2 > 0 && base1 + len1 == base2;
```

```

        // Copy second run into temp array
        Object[] a = this.a; // For performance
        Object[] tmp = ensureCapacity(len2);
        System.arraycopy(a, base2, tmp, 0, len2);

        int cursor1 = base1 + len1 - 1;  // Indexes into a
        int cursor2 = len2 - 1;          // Indexes into tmp array
        int dest = base2 + len2 - 1;     // Indexes into a

        // Move last element of first run and deal with degenerate cases
        a[dest--] = a[cursor1--];
        if (--len1 == 0) {
            System.arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
            return;
        }
        if (len2 == 1) {
            dest -= len1;
            cursor1 -= len1;
            System.arraycopy(a, cursor1 + 1, a, dest + 1, len1);
            a[dest] = tmp[cursor2];
            return;
        }

        int minGallop = this.minGallop;  // Use local variable for performance
    outer:
        while (true) {
            int count1 = 0; // Number of times in a row that first run won
            int count2 = 0; // Number of times in a row that second run won

            /*
             * Do the straightforward thing until (if ever) one run
             * appears to win consistently.
             */
            do {
                if (DEBUG) assert len1 > 0 && len2 > 1;
                if (((Comparable) tmp[cursor2]).compareTo(a[cursor1]) < 0) {
                    a[dest--] = a[cursor1--];
                    count1++;
                    count2 = 0;
                    if (--len1 == 0)
                        break outer;
                } else {
                    a[dest--] = tmp[cursor2--];
                    count2++;
                    count1 = 0;
                    if (--len2 == 1)
                        break outer;
                }
            } while ((count1 | count2) < minGallop);

            /*
             * One run is winning so consistently that galloping may be a
             * huge win. So try that, and continue galloping until (if ever)
```

```
             * neither run appears to be winning consistently anymore.
             */
            do {
                if (DEBUG) assert len1 > 0 && len2 > 1;
                count1 = len1 - gallopRight((Comparable) tmp[cursor2],
                a, base1, len1, len1 - 1);
                if (count1 != 0) {
                    dest -= count1;
                    cursor1 -= count1;
                    len1 -= count1;
                    System.arraycopy(a, cursor1 + 1, a, dest + 1,
                    count1);
                    if (len1 == 0)
                        break outer;
                }
                a[dest--] = tmp[cursor2--];
                if (--len2 == 1)
                    break outer;

                count2 = len2 - gallopLeft((Comparable) a[cursor1], tmp,
                0, len2, len2 - 1);
                if (count2 != 0) {
                    dest -= count2;
                    cursor2 -= count2;
                    len2 -= count2;
                    System.arraycopy(tmp, cursor2 + 1, a, dest + 1,
                    count2);
                    if (len2 <= 1)
                        break outer; // len2 == 1 || len2 == 0
                }
                a[dest--] = a[cursor1--];
                if (--len1 == 0)
                    break outer;
                minGallop--;
            } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
            if (minGallop < 0)
                minGallop = 0;
            minGallop += 2;  // Penalize for leaving gallop mode
        }  // End of "outer" loop
        this.minGallop = minGallop < 1 ? 1 : minGallop;  // Write back
        to field

        if (len2 == 1) {
            if (DEBUG) assert len1 > 0;
            dest -= len1;
            cursor1 -= len1;
            System.arraycopy(a, cursor1 + 1, a, dest + 1, len1);
            a[dest] = tmp[cursor2];  // Move first elt of run2 to front
            of merge
        } else if (len2 == 0) {
            throw new IllegalArgumentException(
                "Comparison method violates its general contract!");
        } else {
            if (DEBUG) assert len1 == 0;
            if (DEBUG) assert len2 > 0;
            System.arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
        }
    }

    /**
```

```
     * Ensures that the external array tmp has at least the specified
     * number of elements, increasing its size if necessary.  The size
     * increases exponentially to ensure amortized linear time complexity.
     *
     * @param minCapacity the minimum required capacity of the tmp array
     * @return tmp, whether or not it grew
     */
    private Object[]  ensureCapacity(int minCapacity) {
        if (tmp.length < minCapacity) {
            // Compute smallest power of 2 > minCapacity
            int newSize = minCapacity;
            newSize |= newSize >> 1;
            newSize |= newSize >> 2;
            newSize |= newSize >> 4;
            newSize |= newSize >> 8;
            newSize |= newSize >> 16;
            newSize++;

            if (newSize < 0) // Not bloody likely!
                newSize = minCapacity;
            else
                newSize = Math.min(newSize, a.length >>> 1);

            @SuppressWarnings({"unchecked", "UnnecessaryLocalVariable"})
            Object[] newArray = new Object[newSize];
            tmp = newArray;
        }
        return tmp;
    }

    /**
     * Checks that fromIndex and toIndex are in range, and throws an
     * appropriate exception if they aren't.
     *
     * @param arrayLen the length of the array
     * @param fromIndex the index of the first element of the range
     * @param toIndex the index after the last element of the range
     * @throws IllegalArgumentException if fromIndex > toIndex
     * @throws ArrayIndexOutOfBoundsException if fromIndex < 0
     *         or toIndex > arrayLen
     */
    private static void rangeCheck(int arrayLen, int fromIndex, int toIndex) {
        if (fromIndex > toIndex)
            throw new IllegalArgumentException("fromIndex(" + fromIndex +
                       ") > toIndex(" + toIndex+")");
        if (fromIndex < 0)
            throw new ArrayIndexOutOfBoundsException(fromIndex);
        if (toIndex > arrayLen)
            throw new ArrayIndexOutOfBoundsException(toIndex);
    }
}
```